The Servers of Serverless Computing: A Formal Revisitation of Functions as a Service

Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Davide Sangiorgi, Stefano Zingaro

The Servers of Serverless Computing: A Formal Revisitation of Functions as a Service

Saverio Giallorenzo
University of Southern Denmark, Odense, Denmark (former)
University of Bologna/INRIA, Italy (current)
saverio.giallorenzo@gmail.com

Ivan Lanese
University of Bologna/INRIA, Italy
ivan.lanese@gmail.com

Fabrizio Montesi
University of Southern Denmark, Odense, Denmark
fmontesi@imada.sdu.dk

Davide Sangiorgi
University of Bologna/INRIA, Italy
davide.sangiorgi@unibo.it

Stefano Pio Zingaro
University of Bologna/INRIA, Italy
stefanopio.zingaro@unibo.it

Abstract

Serverless computing is a paradigm for programming cloud applications in terms of stateless functions, executed and scaled in proportion to inbound requests. Here, we revisit SKC, a calculus capturing the essential features of serverless programming. By exploring the design space of the language, we refined the integration between the fundamental features of the two calculi that inspire SKC: the λ- and the π-calculus. That investigation led us to a revised syntax and semantics, which support an increase in the expressiveness of the language. In particular, now function names are first-class citizens and can be passed around. To illustrate the new features, we present step-by-step examples and two non-trivial use cases from artificial intelligence, which model, respectively, a perceptron and an image tagging system into compositions of serverless functions. We also illustrate how SKC supports reasoning on serverless implementations, i.e., the underlying network of communicating, concurrent, and mobile processes which execute serverless functions in the cloud. To that aim, we show an encoding from SKC to the asynchronous π-calculus and prove it correct in terms of an operational correspondence.

Dedicated to Maurizio Gabbrielli, on his 60th birthday
(... rob da mët !)

2012 ACM Subject Classification Computer systems organization → Cloud computing; Theory of computation → Concurrency

Keywords and phrases Serverless computing, Process calculi, Pi-calculus

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.5

Funding *Fabrizio Montesi*: Partially supported by Villum Fonden (grant no. 29518).
Davide Sangiorgi: Partially supported by MIUR-PRIN project "Analysis of Program Analyses" (ASPRA, ID 201784YSZ5_004).

1 Introduction

Background

Serverless computing, or Functions as a Service (FaaS), is a recent paradigm for programming cloud applications [2]. Programmers code applications in terms of functions. Function definitions can be stored inside of dedicated repositories, and their execution can be triggered by events. An underlying cloud framework is responsible for executing functions whenever triggered, by utilising efficiently a pool of servers. Asynchronous interaction is prominent in this setting, since a function might trigger the execution of another function on another server, and then fetch the result later on to perform further computation.

A framework for serverless computing consists of two layers: a *language layer*, which programmers use to code their applications in terms of (asynchronous) functions; and an *implementation layer*, which executes applications written in the language layer by leveraging a distributed system (the cloud). One can see the first layer as the frontend for developers, and the second layer as the backend that makes serverless systems tick.

Both the language and implementation layers of serverless are complex. Different vendors implement them in different ways with different limitations, in particular regarding how programs can be composed and reasoned about [2, 21, 17]. This motivated recent studies on formal "core" languages for serverless computing, which aim at providing solid foundations for developing and reasoning on serverless systems [13, 20]. We call such core languages *serverless calculi*.

Serverless calculi are based on the λ-calculus: the language layer of serverless frameworks deals with functions, and the λ-calculus is the undisputed reference model for functional programming. The Serverless Kernel Calculus (SKC, or "sketch") is a concurrent λ-calculus enriched with asynchronous function evaluation, futures, and a stateful function repository [13]. SKC leverages the call-by-value evaluation strategy for the λ-calculus to provide a uniform treatment of substitutions for both local and remote serverless computations: evaluating a function synchronously or asynchronously leads to the same substitution carrying the result to the caller.

Motivation

Our work originates from two main motivations.

The first motivation regards the design of the language layer. Both calculi from [13, 20] enhance the λ-calculus with features from process calculi to represent concurrent execution of functions. However, some of these features are slightly different and sometimes more limited. For example, the calculus in [20] uses a "fresh name" condition to transfer the result of function evaluations instead of an explicit operator for name scoping, like name restriction in the π-calculus [27]. SKC adopts a restriction operator to create private names of futures, which are used for the same purpose. Yet, it does not support the creation of new function names that can be stored in the shared function repository, so it is not possible to create private function definitions or private stored data. Furthermore, the expressiveness of SKC is only briefly discussed in [13].

Our second motivation regards the implementation layer. Implementing a serverless framework requires dealing with communication and mobility, in the sense that the connections among the underlying cloud servers and the concurrent processes that run within them change at runtime. The reference theory for mobile processes is the π-calculus, but to the best of our knowledge there is still no exploration of how serverless calculi can be formally linked to it.

This article

In this article, we revisit the theory of SKC and provide a more extensive presentation of its features.

Our new version of SKC (Section 2) provides a few improvements, which are given by a better integration of the essential features of the λ-calculus and the π-calculus. Specifically,

- functions can now be parametric on the names of other functions available in a serverless system, whereas before all references to functions in the repository were statically fixed;
- it is now possible to create new function names for the repository dynamically, so the repository of available functions can now grow freely at runtime.

These new features enhance the expressiveness of the language, which we illustrate through small examples (Section 2) and two use cases (Section 3) from artificial intelligence, one implementing the perceptron algorithm and one for distributed tagging of large images.

We present two semantic interpretations for our version of SKC. The first (Section 2) is a refinement of the original reduction semantics from [13], which supports the aforementioned improvements. This high-level semantics is intended for developers to reason abstractly about SKC programs. The second semantic interpretation is a formalisation of a possible implementation layer for SKC, given in terms of an encoding (Section 5) from SKC to the asynchronous π-calculus [41] (recalled in Section 4). The encoding is inspired by Milner's encoding of the call-by-value λ-calculus [26]. It shows how serverless functions can be implemented by servers (replicated processes in the π-calculus) that can be triggered by messages from clients, and how a serverless implementation layer can be modelled in terms of communications among processes. We prove that the encoding is correct in terms of an operational correspondence result.

Our results show that standard techniques from process calculi can be useful to understand the two layers of serverless calculi. Hopefully, this understanding could also provide foundations for tackling some outstanding questions in serverless computing. For example, predicting resource usage and costs is challenging in general, since it requires knowing how functions are executed by the implementation layer.

2 The Serverless Kernel Calculus, Revised

We now present our refined version of the Serverless Kernel Calculus (SKC).

Configurations	C	$::=$	$\langle S, \mathcal{D} \rangle \mid \boldsymbol{\nu} n\, C$
Definition repository	$\mathcal{D}$	$::=$	$\{(f_1, M_1), \ldots, (f_k, M_k)\} \quad (k \geq 0)$
Systems	S, S'	$::=$	$c \blacktriangleleft M \mid S \mid S' \mid \boldsymbol{\nu} n\, S \mid 0$
Functions	M, N	$::=$	$M\ N \mid V \mid$
			$\mathsf{call}\ h \mid \mathsf{store}\ h\ N\ M \mid \mathsf{take}\ h \mid \boldsymbol{\nu} f\ M \mid \mathsf{async}\ M \mid c$
Values	V, V'	$::=$	$x \mid \lambda x.\, M \mid f$
Restrictable names	n	$::=$	$c \mid f$
	h	$::=$	$f \mid x$
Function names	f	$\in$	Fun
Future names	c	$\in$	Fut
Variables	x	$\in$	Var

Figure 1 Syntax of SKC.

2.1 Syntax

The syntax of SKC terms is given in Figure 1, and described in the following.

We assume three disjoint enumerable sets of *names*: function names, ranged over by f; future names, ranged over by c; and the usual variables of λ-calculus, ranged over by x.

Configurations and definition repositories

A configuration C represents a running serverless architecture. In a configuration of the form $\langle S, \mathcal{D} \rangle$:

- S is a system composed of functions that are currently being evaluated; and
- $\mathcal{D}$ is a repository of function definitions that can be triggered (and updated) at runtime.

We treat definition repositories $\mathcal{D}$ as partial maps from function names (ranged over by f) to function bodies (ranged over by M). Thus, for example, the writings $\mathcal{D}(f) = M$ and $(f, M) \in \mathcal{D}$ are equivalent. Definition repositories are mutable: their content can change at runtime, as we are going to see when we discuss the syntax of functions.

The configuration term $\boldsymbol{\nu} n\, C$ restricts the scope of a name n to C, binding n in C. A restrictable name n can be either a function name f or a future name c.

Systems

A system S is a composition of functions that are being evaluated in parallel.

Term $c \blacktriangleleft M$ represents a function under evaluation, whose result will be made available under the future name c upon termination.

Systems of running functions can be composed in parallel, written $S \mid S'$. Term 0 is the unit of parallel composition.

Names of futures and functions can be restricted in systems as well, using the term $\boldsymbol{\nu} n\, S$.

The restriction operator ν binds stronger than the parallel operator $\mid$. Thus, for example, $\boldsymbol{\nu} n\, S \mid S'$ is interpreted as $(\boldsymbol{\nu} n\, S) \mid S'$.

Functions

The language of functions includes the usual terms of λ-calculus: the variable term x, the application term MN, and the abstraction term $\lambda x.\, M$. We distinguish the syntactic category of values (V) to make the presentation of our call-by-value semantics easier later on.

We extend the usual syntax of functions with terms for using the definition repository and futures.

A function f in the definition repository can be invoked by term $\mathsf{call}\ f$, which abstractly represents the "triggering" of a function in the definition repository by an event. The syntax is actually more general: in term $\mathsf{call}\ h$, h can be either a function name or a variable. This enables abstracting over function names, as in $\lambda x.\, \mathsf{call}\ x$. Passing a function name as an argument is enabled since term f is a value.

The primitives store and take manipulate the definition repository. Specifically, term $\mathsf{store}\ h\ N\ M$ updates the definition repository with the mapping (h, N) – h is now mapped to N – and then proceeds by evaluating M. Dually, term $\mathsf{take}\ h$ removes the function with name h from the definition repository and then proceeds as its body. For example, assuming that the definition repository contains a mapping (f, M), then $\mathsf{take}\ f$ would erase that mapping and proceed as M.

We allow for function names to be restricted, written $\boldsymbol{\nu} f\, M$, which allows for the definition of functions in the repository that have "private names".

Moving to futures, term $\mathsf{async}\ M$ starts the asynchronous execution of M. The idea is that M is going to run in parallel, and that this execution is going to be connected to the original term $\mathsf{async}\ M$ through a fresh future name that is created automatically. For example, a system running $c' \blacktriangleleft \mathsf{async}\ M$ would reduce to $\boldsymbol{\nu} c\,(c' \blacktriangleleft c \mid c \blacktriangleleft M)$. Term c represents a function waiting for a future to be resolved: it will be replaced with the result of the parallel running function $c \blacktriangleleft M$ once M produces a value that can be returned. Note that futures c are not values, hence they cannot be, e.g., passed to functions. One may increase flexibility e.g., by "thunking" futures such that $\mathsf{async}\ M$ becomes the value $\lambda x.\, c$ instead of the non-value c. We leave the exploration of this direction for future work.

Equality and substitution

The ν and λ operators bind names, giving rise to the expected notions of free names ($\mathtt{fn}(-)$) and bound names ($\mathtt{bn}(-)$). Names ($\mathtt{n}(-)$) are the union of free and bound names. Thus, we obtain the usual notions of α-equivalence, written $=_\alpha$, and capture-avoiding substitution for functions, written $M\{V/x\}$ (read "V replaces x in M").

In the remainder, we equate α-equivalent systems, and the same for functions. Consistently with viewing definition repositories as maps, equality of definition repositories allows for swap – the order in which definitions are given inside of $\mathcal{D}$ does not matter. This is equivalent to extensional equality over finite maps: $\mathcal{D} = \mathcal{D}'$ if and only if $\mathcal{D}$ and $\mathcal{D}'$ have the same domain of definition $F \subset \mathsf{Fun}$ and $\mathcal{D}(f) = \mathcal{D}'(f)$ for all $f \in F$.

2.2 Semantics

We present now the semantics of SKC, given in terms of structural equivalence of terms and a reduction relation that captures term dynamics.

Both structural equivalence and the reduction relation make use of the auxiliary definition of evaluation contexts, given in Figure 2. An evaluation context $\mathcal{E}$ is a running function with a hole $[\cdot]$ that can be replaced with a function term M. We write $\mathcal{E}[M]$ for the term obtained by replacing the hole in $\mathcal{E}$ with M.

$$\begin{array}{lcl} \mathcal{E} & ::= & c \blacktriangleleft \mathcal{E}_\lambda \\ \mathcal{E}_\lambda & ::= & [\cdot] \mid \lambda x.\, M\ \mathcal{E}_\lambda \mid \mathcal{E}_\lambda\ M \end{array}$$

Figure 2 SKC, evaluation contexts.

Structural equivalence

Terms in SKC give rise to the expected equivalences regarding name scoping and parallel composition. This is formalised by the *structural equivalence* $\equiv$ between terms, which is the smallest congruence on SKC terms satisfying the rules in Figure 3.

These rules are straightforward adaptations of the typical rules for scoping and the parallel operator found in process calculi. The first row of rules axiomatise that parallel composition ($\mid$) behaves as a commutative monoid with 0 as identity element. The second row deals with extrusion of function names in evaluation contexts. The third row contains rules for garbage collection of restrictions, swapping of restrictions, and name extrusion in systems and configurations.

$$\frac{}{S \mid S' \equiv S' \mid S} \qquad \frac{}{S \mid (S' \mid S'') \equiv (S \mid S') \mid S''} \qquad \frac{}{S \mid 0 \equiv S}$$

$$\frac{f \notin \mathtt{fn}(\mathcal{E})}{\boldsymbol{\nu} f\ \mathcal{E}[M] \equiv \mathcal{E}[\boldsymbol{\nu} f\ M]} \qquad \frac{f \notin \mathtt{fn}(\mathcal{E}_\lambda)}{\boldsymbol{\nu} f\ \mathcal{E}_\lambda[M] \equiv \mathcal{E}_\lambda[\boldsymbol{\nu} f\ M]}$$

$$\frac{}{\boldsymbol{\nu} n\ 0 \equiv 0} \qquad \frac{}{\boldsymbol{\nu} n\ \boldsymbol{\nu} n'\ S \equiv \boldsymbol{\nu} n'\ \boldsymbol{\nu} n\ S} \qquad \frac{n \notin \mathtt{fn}(S')}{\boldsymbol{\nu} n\ (S \mid S') \equiv \boldsymbol{\nu} n\ S \mid S'} \qquad \frac{n \notin \mathtt{fn}(\mathcal{D})}{\boldsymbol{\nu} n\ \langle S, \mathcal{D}\rangle \equiv \langle \boldsymbol{\nu} n\ S, \mathcal{D}\rangle}$$

Figure 3 SKC, rules for structural congruence.

$$\frac{}{\langle \mathcal{E}[(\lambda x.\ M)\ V], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[M\{V/x\}], \mathcal{D}\rangle}\ \beta$$

$$\frac{c \notin \mathtt{fn}(M) \quad c \notin \mathtt{n}(\mathcal{E})}{\langle \mathcal{E}[\mathsf{async}\ M], \mathcal{D}\rangle \longrightarrow \langle \boldsymbol{\nu} c\ (\mathcal{E}[c] \mid c \blacktriangleleft M), \mathcal{D}\rangle}\ \textsc{async}$$

$$\frac{}{\langle \mathcal{E}[c] \mid c \blacktriangleleft V, \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[V] \mid c \blacktriangleleft V, \mathcal{D}\rangle}\ \textsc{push}$$

$$\frac{}{\langle \mathcal{E}[\mathsf{store}\ f\ N\ M], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[M], \mathcal{D}[f \mapsto N]\rangle}\ \textsc{store}$$

$$\frac{}{\langle \mathcal{E}[\mathsf{call}\ f], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[\mathcal{D}(f)], \mathcal{D}\rangle}\ \textsc{call} \qquad \frac{}{\langle \mathcal{E}[\mathsf{take}\ f], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[\mathcal{D}(f)], \mathtt{undef}(\mathcal{D}, f)\rangle}\ \textsc{take}$$

$$\frac{\langle S, \mathcal{D}\rangle \longrightarrow \langle S', \mathcal{D}'\rangle}{\langle \boldsymbol{\nu} n\ S, \mathcal{D}\rangle \longrightarrow \langle \boldsymbol{\nu} n\ S', \mathcal{D}'\rangle}\ \textsc{res-s} \qquad \frac{\langle S_1, \mathcal{D}\rangle \longrightarrow \langle S_1', \mathcal{D}'\rangle}{\langle S_1 \mid S_2, \mathcal{D}\rangle \longrightarrow \langle S_1' \mid S_2, \mathcal{D}'\rangle}\ \textsc{par}$$

$$\frac{C \longrightarrow C'}{\boldsymbol{\nu} n\ C \longrightarrow \boldsymbol{\nu} n\ C'}\ \textsc{res-c} \qquad \frac{C_1 \equiv C_1' \quad C_1' \longrightarrow C_2' \quad C_2' \equiv C_2}{C_1 \longrightarrow C_2}\ \textsc{str}$$

Figure 4 SKC, reduction rules.

Reductions

We can now define the reduction relation $\longrightarrow$, which formalises the execution of terms in SKC. Relation $\longrightarrow$ is the smallest relation closed under the rules displayed in Figure 4.

The semantics of SKC is based on the call-by-value evaluation strategy for λ-calculus. Specifically, rule β allows for an application to reduce only when its argument is a value (V).

Rule ASYNC models the asynchronous execution of a function M: it creates a new future c, a parallel running function to compute the result of M in c, and binds c to the parallel composition of the caller (which is now waiting to receive the result) and the new running function. When the created running function reduces to a result value, the caller can collect this result by rule PUSH.

In rule STORE, a term $\mathsf{store}\ f\ N\ M$ updates the definition repository $\mathcal{D}$ with the mapping (f, N). The notation $\mathcal{D}[f \mapsto N]$ means that $\mathcal{D}$ is updated to contain the mapping (f, N): if f was already mapped to something, the old mapping is discarded.

In rule CALL, a term $\mathsf{call}\ f$ retrieves the body of the function f from the definition repository, if it is defined, and runs it. A term $\mathsf{call}\ f$ is stuck if the definition repository does

not contain any mapping for f (but can become unstuck if a mapping appears later on). Rule TAKE is similar, but the mapping for the called function is removed from the definition repository: we write $\texttt{undef}(\mathcal{D}, f)$ for the repository obtained by removing the mapping for f from $\mathcal{D}$.

The other rules are the expected ones for dealing with restriction (RES-S and RES-C), parallel composition (PAR), and structural equivalence (STR): reductions under restriction and parallel composition can be lifted, and the reduction relation $\longrightarrow$ is closed under the structural equivalence $\equiv$.

▶ **Example 1** (Local vs Async execution)**.** As we are going to see in Section 3, the definition repository is useful to store data and functions that are commonly reused. By itself, term call f retrieves the body of function f from the repository and runs it locally. One can combine call with async to execute the retrieved function asynchronously, which gives some control on how functions from the repository should be executed.

The caller of a function does not need to worry about which strategy is used by the callee, since the semantics of SKC makes both to eventually reduce to the same result. For example, assume that $\mathcal{D}(f) = V$. The following reduction chains show the respective behaviours of the two strategies.

$$\langle c \blacktriangleleft \mathsf{call}\ f, \mathcal{D}\rangle \longrightarrow \langle c \blacktriangleleft V, \mathcal{D}\rangle \tag{1}$$

$$\begin{aligned}&\langle c \blacktriangleleft \mathsf{async\ call}\ f, \mathcal{D}\rangle\\ &\longrightarrow \langle \boldsymbol{\nu} c'\ (c \blacktriangleleft c' \mid c' \blacktriangleleft \mathsf{call}\ f), \mathcal{D}\rangle\\ &\longrightarrow \langle \boldsymbol{\nu} c'\ (c \blacktriangleleft c' \mid c' \blacktriangleleft V), \mathcal{D}\rangle\\ &\longrightarrow \langle \boldsymbol{\nu} c'\ (c \blacktriangleleft V \mid c' \blacktriangleleft V), \mathcal{D}\rangle\end{aligned} \tag{2}$$

The resulting term has the same behaviour of the one resulting from the local execution. One could make the two terms syntactically equal by implementing garbage collection for unused futures (and the related values).

▶ **Example 2** (Shared state)**.** The definition repository can be used to store and share state. A simple example is keeping a counter of requests. We abuse notation and use arithmetic operators and natural numbers in SKC – as presented in Section 3. The counter can be initialised with

$$(\mathsf{store}\ \mathit{counter}\ V_0\ M)$$

where V_0 is the initial value, and incremented with

$$(\lambda x.\,\mathsf{store}\ \mathit{counter}\ \ (\mathsf{call}\ \mathit{sum}\ 1\ x)\ \ M)\,(\mathsf{take}\ \mathit{counter})$$

In both the cases M is a continuation.

▶ **Example 3** (Libraries)**.** Updating shared state as in the previous example happens often in serverless computing. One could think of offering a replace primitive as syntactic sugar.

$$\mathsf{replace}\ h\ N\ M \triangleq (\lambda x.\,\mathsf{store}\ h\ \ (N\ x)\ \ M)\,(\mathsf{take}\ h)$$

We can then rewrite our previous counter example as follows.

$$\mathsf{replace}\ \mathit{counter}\ \ (\mathsf{call}\ \mathit{sum}\ 1)\ \ M$$

We can actually use the definition repository of SKC to offer these kinds of user-defined functions as libraries. For example, the replace function above can be put in a repository and then invoked by programs as follows.

$$\langle\ c \blacktriangleleft (\mathsf{call}\ \mathit{replace})\ \ \mathit{counter}\ \ (\mathsf{call}\ \mathit{sum}\ 1)\ \ M,\ \{(\mathit{replace}, \lambda \mathit{name}.\, \lambda \mathit{op}.\, \lambda \mathit{cont}.\, (\lambda x.\, \mathsf{store}\ \mathit{name}\ \ (\mathit{op}\ x)\ \ \mathit{cont})\, (\mathsf{take}\ \mathit{name}))\}\ \rangle$$

▶ **Example 4** (Private state)**.** By combining the restriction operator with the primitives for manipulating the definition repository, we obtain private storage. This can be used, for example, to implement a log in a composition of function calls.

The library function *newLog* below creates a private log and initialises it with an empty list (*nil*, cf. Section 3).

$$(\mathit{newLog}, \boldsymbol{\nu} \mathit{log}\, (\mathsf{store}\ \mathit{log}\ \mathit{nil}\ \mathit{log}))$$

A function can now create a log and pass it to other functions that it invokes, as follows (we omit the concrete composed functions M and N). We assume the existence of a library function *pair* that takes two arguments and returns a pair (we show the definition of *pair* in Section 3).

$$(\lambda x.\, (\mathsf{call}\ \mathit{pair}\, ((M\ x)(N\ x)))\, x)\, (\mathsf{call}\ \mathit{newLog})$$

The functions M and N can internally use the library function *replace* to update the log by appending messages to the list. The idea is that $N\ x$ is evaluated first, the result of which is then taken by M along with x (the name of the private log). The function *pair* retrieved from the repository then takes the final result from M and the x as the elements of the pair.

Notice that if the programmer wishes to run M and/or N asynchronously, they can just prepend the inner calls to these functions with the async keyword.

Using similar devices, one can implement stack traces and other tracing mechanisms with different scoping rules (per session, per client, etc.).

3 Use Cases

In this section, we present a couple of non-trivial SKC programs as use cases to show and comment how developers can use SKC to reason about their serverless implementations.

For the use cases, we take inspiration from the context of artificial intelligence (specifically, machine learning), which can benefit considerably from the dynamic scalability of serverless architectures [17, 21].

First, we present an implementation of the perceptron [34, 12], an algorithm for binary classification which can decide to which of two classes an input, represented by a vector of numbers, belongs. Then, we present a scatter-gather algorithm that uses a neural network trained for image classification to infer the semantic content of an input image.

Before presenting the examples, we assume to extend SKC with the syntax for conditionals `if` M `then` M' `else` M'', with the traditional semantics. Moreover, we assume that the definition repository $\mathcal{D}$ executing our examples includes the definition of the Church encodings for pairs and lists.

$$\begin{array}{lcllcl}
\mathcal{D}(pair) & = & \lambda x.\,\lambda y.\,\lambda z.\,z\ x\ y & \mathcal{D}(isNil) & = & \mathsf{call}\ first \\
\mathcal{D}(first) & = & \lambda p.\,p(\lambda x.\,\lambda y.\,x) & \mathcal{D}(cons) & = & \lambda h.\,\lambda t.\,\mathsf{call}\ pair\ \mathtt{false}\ (\mathsf{call}\ pair\ h\ t) \\
\mathcal{D}(second) & = & \lambda p.\,p(\lambda x.\,\lambda y.\,y) & \mathcal{D}(head) & = & \lambda z.\,\mathsf{call}\ first\ (\mathsf{call}\ second\ z) \\
\mathcal{D}(nil) & = & \mathsf{call}\ pair\ \mathtt{true}\ \mathtt{true} & \mathcal{D}(tail) & = & \lambda z.\,\mathsf{call}\ second\ (\mathsf{call}\ second\ z)
\end{array}$$

Hence, for example, we can write the list $1, 2, 3$ as $(\mathsf{call}\ cons\ 1\ (\mathsf{call}\ cons\ 2\ (\mathsf{call}\ cons\ 3\ \mathsf{call}\ nil)))$.

We also assume to extend SKC with the standard arithmetic $(+, -, *)$ and relational $(>, =)$ operators, on which we build the following functions, also assumed present in the definition repository $\mathcal{D}$.

$$\begin{array}{lcllcllcl}
\mathcal{D}(sum) & = & \lambda x.\,\lambda y.\,x + y & \mathcal{D}(prod) & = & \lambda x.\,\lambda y.\,x * y & \mathcal{D}(sub) & = & \lambda x.\,\lambda y.\,x - y \\
\mathcal{D}(gt) & = & \lambda x.\,\lambda y.\,x > y & \mathcal{D}(eq) & = & \lambda x.\,\lambda y.\,x = y & & &
\end{array}$$

Finally, we extend SKC with *let* expressions: $let\ x \leftarrow M'\ in\ M \triangleq (\lambda x.\,M)\ M'$.

3.1 A Serverless Perceptron Algorithm

The perceptron, first introduced by Rosenblatt [34], is an algorithm for binary classification that, given an input and a set of weights, produces an output representing the predicted class. A simple application of Rosenblatt's perceptron is the prediction of the logical conjunction. Given the list of inputs $(0, 0)$, $(0, 1)$, $(0, 1)$, and $(1, 1)$, we aim at predicting the target values 0, 0, 0, and 1 respectively.

To obtain a trained model, i.e., a list of weights enabling the perceptron to find the targets corresponding to some given inputs, we start from a zero-initialised vector of weights (a list with the same cardinality of the inputs whose elements are all 0). The *train*ing algorithm consists of an iteration over a set of inputs labelled with the correct classification to return a new list of updated weights, aligned with the new provided "knowledge". Concretely, a training set for the logical conjunction contains pairs like $((0, 1), 0)$ and $((1, 1), 1)$, where the left element is the list of input's features and the right one is the classifying label – in this case, the binary representation of the truth value of the conjunction of the two elements in the list. Once trained, the weights can be used to *predict* the classification of further inputs. To complete the scenario, both the *predict* and *train*ing functions take a *bias* parameter that, in the perceptron algorithm, acts as the y-intercept of the line that separates the feature space. Without this added parameter it would not be possible to find a solution for the classification problem.

We start by describing the *predict* function, defined in Figure 5, which the *train*ing function uses to calculate an adjustment gradient and update the weights. Mathematically, the classification is based on the formula $comp(f, w, bias) = bias + \sum_{i=0}^{|f|} w_i * f_i$: the item is classified with label "1" if the result is positive, "0" otherwise.

$$\begin{array}{l}
\mathcal{D}(predict) = \ \lambda\, ft.\,\lambda\, ws.\,\lambda\, bias. \\
\quad let\ comp\ \leftarrow (\lambda\, f.\,\lambda\, w.\,\lambda\, \rho. \\
\quad\quad \mathsf{call}\ sum\ (\mathsf{call}\ prod\ (\mathsf{call}\ head\ f)(\mathsf{call}\ head\ w)) \\
\quad\quad\ \mathtt{if}\ (\mathsf{call}\ isNil\ (\mathsf{call}\ tail\ f))\ \mathtt{then}\ 0\ \mathtt{else}\ \mathsf{async}\ \rho\ (\mathsf{call}\ tail\ f)\ (\mathsf{call}\ tail\ w)\ \rho \\
\quad)\ in\ \mathtt{if}\ (\mathsf{call}\ gt\ (\mathsf{call}\ sum\ bias\ (\mathsf{async}\ comp\ ft\ ws\ comp))\ 0)\ \mathtt{then}\ 1\ \mathtt{else}\ 0
\end{array}$$

Figure 5 Function *predict*.

```
D(train) = λ features. λ weights. λ label. λ bias.
  if call eq (async call predict features weights bias) label
  then call pair weights bias
  else
    let routine ← (
      λ f. λ w. λ grad. λ ρ.
        call cons
          call sum (call head w) (call prod grad (call head f))
          if call isNil (call tail f)
            then call nil
            else async ρ (call tail f) (call tail w) grad ρ
      ) in
        let gradient ← (call sub label (async call predict features weights bias))
        in call pair
          async routine features weights gradient routine
          gradient
    )
```

Figure 6 Function *train*.

Given a list of *features* f, a list of *weights* w and a *bias* parameter, the *predict* function calculates the classification of the individual, represented by the list of features, by *sum*ming the *bias* with the recursive *sum* of the pair-wise *prod*ucts of the elements in the two lists. To perform the recursive call, we use the *let*-bound function *comp*. All executions of the *comp* function are asynchronous – called using the async primitive, both inside the definition of *comp* (async ρ ...) and at the initial invocation (async *comp* ...).

The *train*ing function, defined in Figure 6, takes a list of *features*, a list of *weights*, a *label* classifying the individual represented by the features and the *bias* parameter. First, the function (asynchronously) tests whether the weights are already trained to correctly recognise the individual (i.e., the *predict*ion equates the *label*) and, in that case, it returns a pair containing the current *weights* and the *bias*. Otherwise, we define a *let*-bound *routine* function that performs the training by recursively adjusting the weights (w) with respect to the features (f) and a *grad*ient parameter.

The training corresponds to the application of the following mathematical formula, which returns a list of adjusted weights: $routine(f, w, grad) = [w_0 + grad * f_0, \ldots, w_{|w|} + grad * f_{|f|}]$.

Similarly to the *predict* function described above, we use the async primitive to parallelise the execution of the adjustment of the weights. This is done with the initial asynchronous invocation of the *routine* function (async *routine* ...) and the asynchronous recursive call within the definition *routine* (async ρ ...).

Finally, we first define (through the *let* construct) the *gradient* parameter, which corresponds to the distance (i.e., the *sub*traction) between the *label* and the *predict*ion (also executed asynchronously) and then we return a new *pair* containing the result of the execution of the *routine* function – which returns the adjusted weights – and the calculated *gradient* – representing the new bias of the adjusted weights.

▶ **Example 5** (Training and Prediction)**.** Now that we have defined both the *predict* and *train* functions, we can use them to "emulate" the logical conjunction (where the weights 0 and 1 represent false and true values). First, we introduce in the definition repository the utility

function *trainAndStore* which, given the weights and bias of the model – stored as a pair by a function *wsName* in the definition repository –, some *features*, and a *label*, performs the training of the model and replaces the "old" weights and bias with the new, trained ones.

$$\mathcal{D}(\mathit{trainAndStore}) = \lambda\, \mathit{wsName}.\, \lambda\, \mathit{fs}.\, \lambda\, \mathit{label}.\, (\\ \lambda\, \mathit{ws}.\, (\mathsf{store}\ \mathit{wsName}\ (\mathsf{call}\ \mathit{train}\ \mathit{fs}\ (\mathsf{call}\ \mathit{first}\ \mathit{ws})\ \mathit{label}\ (\mathsf{call}\ \mathit{second}\ \mathit{ws}))\ ())\ \mathsf{take}\ \mathit{wsName})$$

To ensure atomicity, *trainAndStore* uses the take primitive to remove the "weights" from the definition environment while it is computing the new ones. In this way, if other clients are trying to access the same weights (name-wise), they are blocked until it finishes computing and it executes the store instruction to "release" them in the definition repository.

We can now train our model to emulate the logical conjunction (*wa* represents the weights and bias of the model) – below we omit $\mathcal{D}$ for compactness.

$$\begin{array}{l} c_0 \blacktriangleleft \mathsf{store}\ \mathit{wa}\ (\mathsf{call}\ \mathit{pair}\ (\mathsf{call}\ \mathit{cons}\ 0\ \mathsf{call}\ \mathit{cons}\ 0\ \mathsf{call}\ \mathit{nil})\ 1)\ () \\ \mid c_1 \blacktriangleleft \mathsf{call}\ \mathit{trainAndStore}\ \mathit{wa}\ (\mathsf{call}\ \mathit{pair}\ (\mathsf{call}\ \mathit{cons}\ 0\ \mathsf{call}\ \mathit{cons}\ 0\ \mathsf{call}\ \mathit{nil})\ 0) \\ \mid c_2 \blacktriangleleft \mathsf{call}\ \mathit{trainAndStore}\ \mathit{wa}\ (\mathsf{call}\ \mathit{pair}\ (\mathsf{call}\ \mathit{cons}\ 0\ \mathsf{call}\ \mathit{cons}\ 1\ \mathsf{call}\ \mathit{nil})\ 0) \\ \mid c_3 \blacktriangleleft \mathsf{call}\ \mathit{trainAndStore}\ \mathit{wa}\ (\mathsf{call}\ \mathit{pair}\ (\mathsf{call}\ \mathit{cons}\ 1\ \mathsf{call}\ \mathit{cons}\ 0\ \mathsf{call}\ \mathit{nil})\ 0) \\ \mid c_4 \blacktriangleleft \mathsf{call}\ \mathit{trainAndStore}\ \mathit{wa}\ (\mathsf{call}\ \mathit{pair}\ (\mathsf{call}\ \mathit{cons}\ 1\ \mathsf{call}\ \mathit{cons}\ 1\ \mathsf{call}\ \mathit{nil})\ 1) \\ \mid c_5 \blacktriangleleft \lambda\, w.\, (\mathsf{call}\ \mathit{predict}\ (\mathsf{call}\ \mathit{cons}\ 0\ \mathsf{call}\ \mathit{cons}\ 1\ \mathsf{call}\ \mathit{nil})\ (\mathsf{call}\ \mathit{first}\ w)\ (\mathsf{call}\ \mathit{second}\ w))\ \mathsf{call}\ \mathit{wa} \end{array}$$

Above, the running function at the bottom (with future c_5) uses the trained weights – at any possible stage of the training, due to the interleaving of the execution – to *predict* the result of the conjunction of the Boolean values 0 and 1.

In the example above, we showed an initial configuration already featuring some running functions. Indeed, since we consider a reduction semantics, we do not model the invocation of functions from outside the system. To consider also the point of view of the user of the serverless system, one could equip SKC with a labelled semantics supporting both the invocation of functions and the retrieval of the results of the evaluation from outside the system. We leave this direction for future work.

3.2 A Serverless Large Image Tagger

In the following, we illustrate the use of the proposed language abstractions in order to model a simple system for tagging large images. The example takes advantage of an Artificial Intelligence (AI) algorithm to extract semantic content from an image. In computer vision it is common practice to segment the content of a photo to assign a label indicating the nature of the object(s) represented in each segment, for example a person, an animal, or a thing. Although modern AI techniques and in particular deep convolutional neural networks are able to predict the semantic content of an image with extreme accuracy, these algorithms normally take small inputs and are not adequate to classify images at ultra-high resolutions, e.g., the recent 4K format corresponding to 3840×2160 pixels. As a reference, MobileNetV2 [35] is a well-known neural network architecture able to achieve fast object classification with accuracy of around 90% over 3-channel colour image inputs of 224×224 pixels.

With the purpose to build a system for annotating ultra-high resolution images, we want to exploit the parallel execution of inference processes to find the labels associated with each image portion and aggregate them at the end of the single computations. The scenario lends itself well to a serverless deployment strategy and can benefit from the language constructs provided by the SKC language. Summarising, our strategy is to split the image into portions that can be quickly annotated by the AI and to aggregate the results computed for each of these parts. This can be simply rendered in SKC by the *tag* function below:

$$\mathcal{D}(tag) = \lambda\, image.\, (\mathsf{call}\ aggregate\ (\mathsf{call}\ split\ image))$$

where function *split* splits the image and function *aggregate* classifies each portion and aggregates the results.

We discuss below in detail function *aggregate*, defined in Figure 7, which scatters the asynchronous computations over the fixed-side portions of the original image (as obtained from the *split* function, omitted here) and assembles their results.

```
D(aggregate) =
  λ splits. (call cons
    call pair
      call first (call head splits)
      async (call infer (call second (call head splits)))
    if (call isNil (call tail splits))
    then nil
    else (async (call aggregate (call tail splits)))
  )
```

Figure 7 Function *aggregate*.

In the snippet in Figure 7, we use the functions to manage pairs and lists already existing in the $\mathcal{D}$ repository, defined at the beginning of this section. A function *infer* is also left undefined in the example. In principle, the *infer* function could be any machine learning algorithm that performs the actual recognition task for the image content given its feature set. Notably, we use function *isNil* from the previous section to calculate the condition of the block if and check whether the result list contains further elements. If not, the algorithm terminates and returns the generated list.

The aggregation function takes a vector of splits, parts of the image, and returns a list containing for each portion its identifier and the list of associated labels. An example of the output for an image representing a seascape in the vicinity of a harbour will take the form ((((0, 0), (223, 223)), (house, sea)), (((0, 224), (224, 447)), (house, boat, sea)), ...). Each identifier contains the coordinates of the portion of the image, represented as two pairs of pixel coordinates. For simplicity, we assume that the result of the *split* function is a list of pairs, the first element of each pair contains the coordinates of the portion while the second element contains the actual portion features.

The *infer* function, that computes the single-portion prediction, retrieves a list of strings, which represents the image tags. The *aggregate* function concatenates the result of the prediction of the first portion of the image with the result of the recursive call on the rest of the split list. Note that both the calls of *infer* and of *aggregate* are made using the async construct. This allows for parallel execution. The result is nevertheless deterministic since the asynchronous functions are independent.

The example presented is intuitive but complex enough to show a highly distributed behaviour. The serverless deployment scenario takes advantage of the constructs for asynchronous execution and helps the programmer to think about the system in a compositional and holistic way.

4 Background: the π-calculus

In this section we introduce the syntax and semantics of π-calculus, and some elements of its theory, to be used in Section 5 to define and prove the correctness of the encoding of SKC into the π-calculus itself. In the encoding, we actually use the *asynchronous* π-calculus, as normal when encoding functional languages, though we usually still call it π-calculus. The asynchronous π-calculus has some striking algebraic properties that fail in the ordinary π-calculus [3], and that will be important in our study. The encoding of a λ-term will be parametric on a name. Such parametric processes are called *abstractions*. The actual instantiation of the parameters of an abstraction F is done via the *application* construct $F\langle\tilde{a}\rangle$. Processes and abstractions form the set of π-*agents* (or simply *agents*). Small letters $a, b, \ldots, x, y, \ldots$ range over the infinite set of names and $\tilde{a}$ denotes a tuple of such names. The grammar of the π-calculus is thus:

$$\begin{array}{rrcl} \text{Agents} & A & := & P \mid F \\ \text{Processes} & P & := & 0 \mid a(\tilde{b}).\,P \mid \overline{a}\langle\tilde{b}\rangle.\,P \mid \boldsymbol{\nu} a\,P \\ & & & \quad \mid P_1 \mid P_2 \mid\ !a(\tilde{b}).\,P \mid F\langle\tilde{a}\rangle \\ \text{Abstractions} & F & := & (\tilde{a})\ P \end{array}$$

In the grammar for processes, 0 is the inactive process. An input-prefixed process $a(\tilde{b}).\,P$, where $\tilde{b}$ has pairwise distinct components, waits for a tuple of names $\tilde{c}$ to be sent along a and then behaves like $P\{\tilde{c}/\tilde{b}\}$, where $\{\tilde{c}/\tilde{b}\}$ is the simultaneous substitution of names $\tilde{b}$ with names $\tilde{c}$. An output particle $\overline{a}\langle\tilde{b}\rangle$ emits names $\tilde{b}$ at a. Parallel composition allows one to run two processes in parallel. The restriction $\boldsymbol{\nu} a\,P$ makes name a local, or private, to P. A replication $!a(\tilde{x}).\,P$ stands for a countable infinite amount of copies of $a(\tilde{x}).\,P$ in parallel.

When the tuple $\tilde{b}$ is empty, the surrounding brackets in prefixes will be omitted. We abbreviate $\boldsymbol{\nu} a \boldsymbol{\nu} b\,P$ as $(\boldsymbol{\nu} a, b)P$. An input prefix $a(\tilde{b}).\,P$, a restriction $\boldsymbol{\nu} b\,P$, and an abstraction $(\tilde{b})\ P$ are binders for names $\tilde{b}$ and b, respectively, and give rise in the expected way to the definition of *free names* ($\mathtt{fn}(\text{-})$) and *bound names* ($\mathtt{bn}(\text{-})$) of a term or a prefix, and α-conversion.

Since the calculus is polyadic, a type system is needed to avoid disagreements in the arities of the tuples of names carried by a given name and in applications of abstractions. We will not present the typing system, however, because not really essential. A *context* E of π is a π-agent in which some subterms have been replaced by the hole $[\cdot]$; then $E[A]$ is the agent resulting from replacing the hole with the term A. Of course it is assumed that, under a type system, we only relate (e.g., using barbed congruence, described later on) agents obeying the same typing, and then we insert them only in contexts that respect such a typing. We assign parallel composition the lowest precedence among the operators.

The operational semantics of the π-calculus is standard [41] and given in Figure 8. Transitions are of the form $P \xrightarrow{a(\tilde{b})} P$ (an input, $\tilde{b}$ are the bound names of the input prefix that has been fired), $P \xrightarrow{\boldsymbol{\nu}\tilde{d}\,\overline{a}\langle\tilde{b}\rangle} P'$ (an output, where $\tilde{d} \subseteq \tilde{b}$ are private names extruded in the output), and $P \xrightarrow{\tau} P'$ (an internal action). As usual, $\Longrightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$, and $\overset{\mu}{\Longrightarrow}$ is $\Longrightarrow\xrightarrow{\mu}\Longrightarrow$.

The reference behavioural equivalence for π-calculus will be the usual *barbed congruence* [28]. We recall its definition. We write $P \Downarrow_a$ if $P \overset{\mu}{\Longrightarrow} P'$, for some P' and μ is an output action at a. (We make only output observable because this is standard in asynchronous calculi.)

$$\frac{}{a(\tilde{b}).\,P \xrightarrow{a(\tilde{b})} P} \qquad \frac{}{!a(\tilde{b}).\,P \xrightarrow{a(\tilde{b})} !a(\tilde{b}).\,P \mid P} \qquad \frac{}{\overline{a}\langle\tilde{b}\rangle.\,P \xrightarrow{\overline{a}\langle\tilde{b}\rangle} P}$$

$$\frac{P \xrightarrow{\boldsymbol{\nu}\tilde{d}\,\overline{a}\langle\tilde{b}\rangle} P' \quad n \in \tilde{b} - \tilde{d}}{\boldsymbol{\nu} n\,P \xrightarrow{(\boldsymbol{\nu} n,\tilde{d})\overline{a}\langle\tilde{b}\rangle} P} \qquad \frac{P \xrightarrow{\mu} P' \quad n \notin \mu}{\boldsymbol{\nu} n\,P \xrightarrow{\mu} \boldsymbol{\nu} n\,P'} \qquad \frac{P \xrightarrow{a(\tilde{b})} P' \qquad Q \xrightarrow{\boldsymbol{\nu}\tilde{d}\,\overline{a}\langle\tilde{b}'\rangle} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu}\tilde{d}\,(P'\{\tilde{b}'/\tilde{b}\} \mid Q')}$$

$$\frac{P \xrightarrow{\mu} P' \quad \mathtt{bn}(\mu) \cap \mathtt{fn}(Q) = \emptyset}{P \mid Q \xrightarrow{\mu} P' \mid Q} \qquad \frac{P\{\tilde{b}/\tilde{a}\} \xrightarrow{\mu} P'}{((\tilde{a})\ P)\langle\tilde{b}\rangle \xrightarrow{\mu} P'}$$

Figure 8 Labelled Transition Semantics for the π-calculus.

▶ **Definition 6** (Barbed congruence)**.** Barbed bisimilarity *is the largest symmetric relation* $\dot{\simeq}$ *on* π*-calculus processes such that* $P \;\dot{\simeq}\; Q$ *implies:*

1. *If* $P \Longrightarrow P'$ *then there is* Q' *such that* $Q \Longrightarrow Q'$ *and* $P' \;\dot{\simeq}\; Q'$*.*
2. $P \Downarrow_a$ *iff* $Q \Downarrow_a$*.*

Processes P *and* Q *are* barbed congruent*, written* $P \approx Q$*, if for each context* E*, it holds that* $E[P] \;\dot{\simeq}\; E[Q]$*.*

As explained in the description of the encoding, we will need capability types [33] for the π-calculus (or their ω-receptive refinement [38]), so to allow us to use the following replication theorem in the correctness proof of the encoding.

▶ **Theorem 7** (Replication theorem)**.** *Suppose only the output capability of* x *can be communicated (or that* x *is* ω*-receptive); then we have:*

- $\boldsymbol{\nu} x\,(P_1 \mid P_2 \mid !x(\tilde{u}).\,R) \approx \boldsymbol{\nu} x\,(P_1 \mid !x(\tilde{u}).\,R) \mid \boldsymbol{\nu} x\,(P_2 \mid !x(\tilde{u}).\,R)$*;*
- $\boldsymbol{\nu} x\,(!P \mid !x(\tilde{u}).\,R) \approx !\boldsymbol{\nu} x\,(P \mid !x(\tilde{u}).\,R)$*.*

5 Encoding SKC into π-calculus

In this section, based on Milner's encoding of call-by-value λ-calculus [26], we study an encoding of SKC into π-calculus. We follow the investigation of the correctness of the representation of pure functions in π-calculus, in particular [26, 37].

We assume a base type system in SKC that at least distinguishes *function values* such as $\lambda x.\,M$ from *nominal values* such as f (of course, depending on how the type system for SKC is defined, function values and nominal values could be collections of types; we do not go into the details of the types as they would obscure the readability of the encoding and its correctness proofs). Moreover, the type system ensures us that whenever a nominal value f is used with the store construct then the repository is not defined on f (thus the replacement of an element of the repository needs a take and then a store).

The encoding is presented in Figure 9. We have to distinguish the encoding of functions and values, that returns an abstraction, from the encoding of the other syntactic categories, that returns a process. We use $[\![-]\!]$ for the former, and $[\![-]\!]^*$ for the latter.

In the encoding of functions and values, the parameter may be thought of as the *location* of that term. A term that becomes a value signals so at its location name and provides access to the body of the value. Such body is replicated and thus may be copied several times. When the value is a λ-function, it receives two names: (the access to) its value-argument, and the location for the resulting term.

A repository of terms is modelled as the parallel composition ($\prod$ denotes indeed n-ary parallel composition) of the encodings of the individual terms, in which a function name f is used to obtain access to the location of the λ-term referred to by f. The imperative nature of the repository is reflected in the reference-like usage of function names. A repository in which f is assigned to M becomes, in the π-calculus, an output $\overline{f}\langle a\rangle$ where a is a fresh name that gives access to (the location of) M. This also explains the encoding of the constructs take f and call f, in which the current content of f is read (as an input). Only in call f such content is then re-emitted; take f does not, as it is supposed to remove f from the repository. Construct store f N M introduces the appropriate output at f, so to add f to the repository.

The encodings use different kinds of names: *location* names, ranged over by p, q; *value* names, ranged over by x, y, for accessing the body of a value; *function* names, ranged over by f, for accessing functions in the repository; *future* names, ranged over by c; *support* names, ranged over by a, b, used to access terms in the repository. In the encoding, as well as in the syntax of SKC, h ranges over value and function names. We assume a type system in the π-calculus in which these kinds of names are separated. Depending on the types of SKC, these kinds may correspond to collections of π-calculus types (we recall that the encoding of pure untyped λ-calculus can be refined to an encoding of typed λ-calculus [41]). A sorting system, à la Milner [25], may be used to keep track of the arities of the names. However we need at least to impose I/O capabilities [33] to make sure that only the output capability of the value and support names is communicated. This is essential for the correctness results below. The typing could however be more precise, by stipulating that function names should follow the discipline of *reference* names of the π-calculus [18] (precisely, a destructive variant in which at any time at most one output at one of these names is available). Similarly, a more precise typing would set value and support names as *ω-receptive* names [38]. For proving more refined correctness properties on the encoding, or using the encoding to validate program transformations or optimisations (e.g., [39, 40, 9]), such refined types would have to be taken into account.

In the encodings of store and async, a τ prefix is added so to make sure that a reduction in SKC corresponds to at least one reduction in π-calculus. This is also needed for Lemma 8. The encodings are extended to contexts and evaluation contexts, in the expected manner, exploiting the compositionality of the encoding.

Lemma 8 shows that only the translation of evaluation contexts yields evaluation contexts in the π-calculus. An occurrence of a term in a π-calculus expression is *unguarded* if that occurrence is not underneath a prefix.

▶ **Lemma 8.** *For any function context E, the hole of the π-calculus context $[\![E]\!]$ is unguarded iff E is an evaluation context.*

▶ **Lemma 9.** *For all M, V, we have: $[\![(\lambda x.\, M)V]\!] \approx [\![M\{^V/_x\}]\!]$.*

Proof. We follow a case analysis on V. The case when $V = x$ or $V = f$ is easy. When V is an abstraction, we proceed by induction on M, exploiting the Replication Theorem 7 (here, the output capability on names x, y is essential).

The induction is similar to that in the proof of validity of β-reduction for the pure call-by-value λ-calculus [36], the main difference is that there are more cases to consider, however the reasoning is analogous. ◀

▶ **Theorem 10** (Operational correspondence, from SKC to π). *If $C \longrightarrow C'$ then $[\![C]\!]^* \longrightarrow\!\approx [\![C']\!]^*$.*

Encoding of definition repositories

$$[\![D]\!]^* \stackrel{\text{def}}{=} \Pi_{(f,M)\in D} \boldsymbol{\nu} a\, (\overline{f}\langle a\rangle \mid !a(q).\, [\![M]\!]\langle q\rangle)$$

Encoding of configurations

$$\begin{aligned} [\![\langle S, D\rangle]\!]^* &\stackrel{\text{def}}{=} [\![S]\!]^* \mid [\![D]\!]^* \\ [\![\boldsymbol{\nu} n\, C]\!]^* &\stackrel{\text{def}}{=} \boldsymbol{\nu} n\, [\![C]\!]^* \end{aligned}$$

Encoding of systems

$$\begin{aligned} [\![c \blacktriangleleft M]\!]^* &\stackrel{\text{def}}{=} \boldsymbol{\nu} p\, ([\![M]\!]\langle p\rangle \mid p(y).\, !c(z).\, \overline{z}\langle y\rangle) \\ [\![S \mid S']\!]^* &\stackrel{\text{def}}{=} [\![S]\!]^* \mid [\![S']\!]^* \\ [\![\boldsymbol{\nu} c\, S]\!]^* &\stackrel{\text{def}}{=} \boldsymbol{\nu} c\, [\![S]\!]^* \\ [\![0]\!]^* &\stackrel{\text{def}}{=} 0 \end{aligned}$$

Encoding of functions

$$\begin{aligned} [\![MN]\!] &\stackrel{\text{def}}{=} (p)\ \boldsymbol{\nu} q\, \big([\![M]\!]\langle q\rangle \mid q(y).\, \boldsymbol{\nu} r\, ([\![N]\!]\langle r\rangle \mid r(w).\, \overline{y}\langle w, p\rangle)\big) \\ [\![\mathsf{call}\ h]\!] &\stackrel{\text{def}}{=} (p)\ (h(a).\, (\overline{h}\langle a\rangle \mid \overline{a}\langle p\rangle) \\ [\![\mathsf{async}\ M]\!] &\stackrel{\text{def}}{=} (p)\ \tau.\, \boldsymbol{\nu} c\, ([\![c]\!]\langle p\rangle \mid [\![c \blacktriangleleft M]\!]^*) \\ [\![\mathsf{store}\ h\ N\ M]\!] &\stackrel{\text{def}}{=} (p)\ \tau.\, ([\![M]\!]\langle p\rangle \mid \boldsymbol{\nu} a\, (\overline{h}\langle a\rangle \mid !a(q).\, [\![N]\!]\langle q\rangle)) \\ [\![\mathsf{take}\ h]\!] &\stackrel{\text{def}}{=} (p)\ h(y).\, \overline{y}\langle p\rangle \\ [\![\boldsymbol{\nu} f\, M]\!] &\stackrel{\text{def}}{=} (p)\ \boldsymbol{\nu} f\, [\![M]\!]\langle p\rangle \\ [\![c]\!] &\stackrel{\text{def}}{=} (p)\ \overline{c}\langle p\rangle \end{aligned}$$

Encoding of values

$$\begin{aligned} [\![\lambda x.\, M]\!] &\stackrel{\text{def}}{=} (p)\ \boldsymbol{\nu} y\, \overline{p}\langle y\rangle.\, !y(x, q).\, [\![M]\!]\langle q\rangle \\ [\![h]\!] &\stackrel{\text{def}}{=} (p)\ \overline{p}\langle h\rangle \end{aligned}$$

Figure 9 The encoding of SKC into the π-calculus.

Proof. We proceed by rule induction. The inductive part follows from the compositionality of the encoding. For the base case, we make a case analysis on the rule applied.

- Rule β. We use Lemma 9.
- Rule ASYNC. We use the definition of the encoding and Lemma 8.
- Rule PUSH. We have (omitting the store, which does not contribute)

$$\mathcal{E}[c] \mid c \blacktriangleleft V \longrightarrow \mathcal{E}[V] \mid c \blacktriangleleft V$$

We assume V is an abstraction (the case when it is a function name is simpler); we then abbreviate the encoding of an abstraction $\lambda x.\, M$ as

$$[\![\lambda x.\, M]\!]\langle p\rangle = \boldsymbol{\nu} y\, (\overline{p}\langle y\rangle \mid [\![M]\!]^y_{\mathrm{V}})$$

In the encoding, we have, for some p::

$$\begin{array}{rcl}
[\![\mathcal{E}[c] \mid c \blacktriangleleft V]\!]^* & = & [\![\mathcal{E}]\!][\overline{c}\langle p\rangle] \mid (\boldsymbol{\nu} q, y\,)(\overline{q}\langle y\rangle \mid [\![V]\!]^y_{\mathrm{V}} \mid q(y).\, !c(z).\, \overline{z}\langle y\rangle) \\
& \longrightarrow\!\!\equiv & [\![\mathcal{E}]\!][\overline{c}\langle p\rangle] \mid \boldsymbol{\nu} y\, ([\![V]\!]^y_{\mathrm{V}} \mid !c(z).\, \overline{z}\langle y\rangle) \\
& \approx & \boldsymbol{\nu} y\, ([\![\mathcal{E}]\!][\overline{p}\langle y\rangle] \mid [\![V]\!]^y_{\mathrm{V}} \mid !c(z).\, \overline{z}\langle y\rangle) \\
& \approx & [\![\mathcal{E}]\!][\boldsymbol{\nu} y\, (\overline{p}\langle y\rangle \mid [\![V]\!]^y_{\mathrm{V}})] \mid \boldsymbol{\nu} y\, ([\![V]\!]^y_{\mathrm{V}} \mid !c(z).\, \overline{z}\langle y\rangle) \\
& \approx & [\![\mathcal{E}[V]]\!]^* \mid [\![c \blacktriangleleft V]\!]^*
\end{array}$$

where in the second use of $\approx$ we exploit the Replication Theorem 7 (i.e., the output capability constraint on y).

- Rule STORE. The rule is

$$\langle \mathcal{E}[\mathsf{store}\ f\ N\ M], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[M], \mathcal{D}[f \mapsto N]\rangle$$

We have, for some Q that does not contain f, and some p:

$$\begin{array}{rcl}
[\![\langle \mathcal{E}[\mathsf{store}\ f\ N\ M], \mathcal{D}\rangle]\!]^* & = & [\![\mathcal{E}]\!]^*[\tau.\, ([\![M]\!]\langle p\rangle \mid \boldsymbol{\nu} a\, (\overline{f}\langle a\rangle \mid !a(q).\, [\![N]\!]\langle q\rangle))] \mid Q \\
& \longrightarrow & [\![\mathcal{E}]\!]^*[([\![M]\!]\langle p\rangle \mid \boldsymbol{\nu} a\, (\overline{f}\langle a\rangle \mid !a(q).\, [\![N]\!]\langle q\rangle))] \mid Q \\
& \approx & [\![\mathcal{E}]\!]^*[([\![M]\!]\langle p\rangle] \mid \boldsymbol{\nu} a\, (\overline{f}\langle a\rangle \mid !a(q).\, [\![N]\!]\langle q\rangle)) \mid Q \\
& = & [\![\langle \mathcal{E}[M], \mathcal{D}[f \mapsto N]\rangle]\!]^*
\end{array}$$

- Rule CALL. The rule is

$$\langle \mathcal{E}[\mathsf{call}\ f], \mathcal{D}\rangle \longrightarrow \langle \mathcal{E}[M], \mathcal{D}\rangle$$

for $M = \mathcal{D}(f)$. We have, for some Q that does not contain f, and some p:

$$\begin{array}{rcl}
[\![\langle \mathcal{E}[\mathsf{call}\ f], \mathcal{D}\rangle]\!] & = & [\![\mathcal{E}]\!]^*[f(a').\, (\overline{f}\langle a'\rangle \mid \overline{a'}\langle p\rangle)] \mid \boldsymbol{\nu} a\, (\overline{f}\langle a\rangle \mid !a(q).\, [\![M]\!]\langle q\rangle \mid Q \\
& \longrightarrow & \boldsymbol{\nu} a\, ([\![\mathcal{E}]\!]^*[\overline{f}\langle a\rangle \mid \overline{a}\langle p\rangle] \mid !a(q).\, [\![M]\!]\langle q\rangle \mid Q) \\
& \approx & \boldsymbol{\nu} a\, ([\![\mathcal{E}]\!]^*[\overline{f}\langle a\rangle] \mid [\![M]\!]\langle p\rangle \mid !a(q).\, [\![M]\!]\langle q\rangle \mid Q) \\
& \approx & [\![\mathcal{E}]\!]^*[[\![M]\!]\langle p\rangle] \mid \boldsymbol{\nu} a\, (\overline{f}\langle a\rangle \mid !a(q).\, [\![M]\!]\langle q\rangle \mid Q) \\
& = & [\![\langle \mathcal{E}[M], \mathcal{D}\rangle]\!]^*
\end{array}$$

where in the first application of $\approx$ we exploit the τ-insensitiveness property of the name a (equivalently, the Replication Theorem 7 on a).

- The case of rule TAKE is easy. ◀

The converse direction is more delicate because the π-calculus encoding presents a number of administrative reductions, which do not correspond to actual reductions in the source calculus. We overcome the problem following [11], which presents an optimisation of Milner's encoding of application (and hence of the encoding of SKC). The optimised encoding, indicated as $\{\!|-|\!\}$ below, is obtained from the initial one by performing a few (deterministic) reductions, at the price of a more complex definition. These reductions are performed in the clause for application, when at least one of the two involved terms is a value. We only show the definition for the different clauses. In the last clause below it is intended that N is not a value. The three clauses are applied only on applications MN in which both M and N are closed (i.e., they have no free λ-variables); if they are applicable they have priority over the clauses of the initial encoding $[\![-]\!]$. Moreover the first clause has priority over the last one.

$$\begin{array}{rcl}
\{\!|(\lambda x.\, M)(\lambda z.\, N)|\!\} & \stackrel{\text{def}}{=} & (p)\ \boldsymbol{\nu} y\, (!y(x,q).\, \{\!|M|\!\}\langle q\rangle \mid !y'(z,r).\, \{\!|N|\!\}\langle r\rangle \mid \overline{y}\langle y', p\rangle) \\
\{\!|(\lambda x.\, M)h|\!\} & \stackrel{\text{def}}{=} & (p)\ \boldsymbol{\nu} y\, (!y(x,q).\, \{\!|M|\!\}\langle q\rangle \mid \overline{y}\langle h, p\rangle) \\
\{\!|(\lambda x.\, M)N|\!\} & \stackrel{\text{def}}{=} & (p)\ \boldsymbol{\nu} y\, (!y(x,q).\, \{\!|M|\!\}\langle q\rangle \mid \{\!|N|\!\}\langle r\rangle \mid r(w).\, \overline{y}\langle w, p\rangle)
\end{array}$$

The following lemma establishes the correctness of the optimised encoding.

▶ **Lemma 11.** *For any C, we have $[\![C]\!]^* \approx \{\![C]\!\}^*$.*

We state the converse of Theorem 10 on the optimised encoding $\{\![-]\!\}$.

▶ **Theorem 12** (Operational correspondence, from π to SKC)**.** *If $\{\![C]\!\}^* \longrightarrow P$ then there is C' with $C \longrightarrow C'$ and $P \approx [\![C']\!]^*$.*

Proof. Using a case analysis similar to that in the proof of Theorem 10, taking on applications, when applicable, the clauses of the optimised encoding. ◀

Both Theorems 10 and 12 can be extended to multi-step reductions (i.e., relation $\Longrightarrow$ in place of the one-step reduction $\longrightarrow$).

6 Discussion and Conclusion

In this work, we explore the SKC serverless calculus under two aspects: its design space (the language layer) and its usefulness in reasoning on serverless implementations (the implementation layer).

Towards the first aim, we consider features from process calculi to represent the concurrent execution of functions and to better integrate the essential features of the two calculi that inspire SKC: the λ- and the π-calculus. Part of that redesign consists in having functions parametric on the names of other functions. That inclusion allows us to create new function names for the function-definition repository dynamically, so the repository of available functions can grow freely at runtime, as well as holding private function definitions and private stored data. As part of that exploration, we present a refined syntax and semantics for SKC, supporting the aforementioned improvements, and we illustrate how those new features enhance the expressiveness of the language through small examples and two use cases from artificial intelligence.

Then, to illustrate how SKC supports reasoning on the implementation layer, we focus on how to translate SKC-defined architectures (of functions) into a network of communicating, concurrent, and mobile processes, representing the actual network of concurrent processes that run those serverless functions in the cloud. We tackle this second task by presenting an encoding from SKC to the asynchronous π-calculus and proving it correct in terms of an operational correspondence result, using standard techniques – whose side-effect is to cast a good outlook on the affordability of extending results from the literature to SKC.

We remark that the choice of using futures as one of the main building blocks of SKC comes from a careful consideration on the minimality of the language. Indeed, if one might consider named channels (as in CCS/π-calculus [24, 41]) a standard choice of communicating systems, in the case of SKC they would increase the distance between the language and concrete serverless implementations. For example, with channels one could have re-usable, bi-directional communication between functions, which is a feature no serverless implementation currently provides. Moreover, our encoding shows a way to define futures using channels while the opposite is also possible [31].

Regarding related work, the proposal closest to SKC is [20], where a calculus more involved than SKC is presented. It captures the low-level details of current serverless implementations (e.g., cold/warm components, storage, and transactions are primitive features of their model), essentially mixing the language and implementation layers. Contrarily, SKC strives to be a kernel model of serverless computing, with the suggested strategy to reason on implementations via encodings. Another work close to SKC is [31], where the authors introduce a λ-calculus with futures. Since the aim of [31] is to formalise and reason on

a concurrent extension of Standard ML, their calculus is more involved than SKC, as it contains primitive operators (handlers and cells) to capture safe non-deterministic concurrent operations, which we can encode as macros in SKC. An interesting future work is to investigate which results from [20, 31] we can adapt to SKC.

Other future directions of research on SKC include the exploration of guarantees on sequential execution across functions, which compels the investigation of new tools to enforce sequential consistency [23] or serialisability [32] of the transformations of the global state [17]. That challenge can be tackled developing static analysis techniques and type disciplines [19, 1] for SKC. Another direction concerns programming models, which should give to programmers an overview of the overall logic of the distributed functions and capture the loosely-consistent execution model of serverless [17]. Choreographic Programming [29, 7] is a promising candidate for that task, as choreographies are designed to capture the global interactions in distributed systems [22], and recent results [6, 8, 15, 16] confirmed their applicability to microservices [10], a neighbouring domain to that of serverless architectures. Such an approach can also cover Edge and Internet-of-Things scenarios (as targeted by projects like AWS Greengrass[1]), using language abstractions borrowed from the world of microservice systems [14]. The language extensions and results on the encoding presented in this paper are stepping stones for a transformation framework between serverless and microservice architectures. There, the final goal would be to provide an infrastructure where, depending on the application context (e.g., the amount of stateful interactions) and inbound load (steadier traffic benefit the always-on microservices deployment, while serverless is more efficient when considering traffic bursts), users (or automatic optimisation systems) can decide whether to deploy a given architecture (or part of it) as a network of serverless functions or microservices. Indeed, if SKC is the model for serverless and π-calculus-inspired process calculi [30] represent microservices, our encoding is a first result towards a framework for the semantic-preserving transition between the two implementation/deployment paradigms.

A clearer understanding of the implementation layer also provides foundations for tackling some outstanding questions in serverless computing. For example, predicting resource usage and costs is challenging in general, since it requires knowing how functions are executed by the implementation layer. This last one is particularly relevant in the per-usage model of serverless architectures, yet it requires to extend SKC with an explicit notion of time to support quantitative behavioural reasoning for timed systems [5, 4]. A starting point could be to use the encoding into π-calculus to prove termination properties of the source SKC language, combining type techniques from functional and concurrent languages [9].

References

1 Davide Ancona et al. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2-3):95–230, 2016.

2 Ioana Baldini, Paul C. Castro, Kerry Shih-Ping Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. In Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya, editors, *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017. doi:10.1007/978-981-10-5026-8_1.

3 Michele Boreale and Davide Sangiorgi. Some congruence properties for π-calculus bisimilarities. *Theor. Computer Science*, 198:159–176, 1998.

[1] https://aws.amazon.com/greengrass/

4 Tomasz Brengos and Marco Peressotti. A uniform framework for timed automata. In *CONCUR*, volume 59 of *LIPIcs*, pages 26:1–26:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

5 Tomasz Brengos and Marco Peressotti. Behavioural equivalences for timed systems. *Logical Methods in Computer Science*, 15(1), 2019.

6 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013.

7 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. In *FACS*, Lecture Notes in Computer Science, pages 17–35. Springer, 2016.

8 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science*, 13(2), 2017.

9 Romain Demangeon, Daniel Hirschkoff, and Davide Sangiorgi. Termination in impure concurrent languages. In *Proc. 21th Conf. on Concurrency Theory*, volume 6269 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2010.

10 Nicola Dragoni et al. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

11 Adrien Durier, Daniel Hirschkoff, and Davide Sangiorgi. Eager functions as processes. In *33nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*. IEEE Computer Society, 2018.

12 Yoav Freund and Robert E Schapire. Large margin classification using the perceptron algorithm. *Machine learning*, 37(3):277–296, 1999.

13 Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Marco Peressotti, and Stefano Pio Zingaro. No more, no less - A formal model for serverless computing. In *Coordination Models and Languages - 21st International Conference, COORDINATION 2019*, volume 11533 of *Lecture Notes in Computer Science*, pages 148–157. Springer, 2019.

14 Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Stefano Pio Zingaro. A language-based approach for interoperability of IoT platforms. In Tung Bui, editor, *51st Hawaii International Conference on System Sciences, HICSS 2018, Hilton Waikoloa Village, Hawaii, USA, January 3-6, 2018*, pages 1–10. ScholarSpace / AIS Electronic Library (AISeL), 2018. URL: `http://hdl.handle.net/10125/50603`.

15 Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli. Applied choreographies. In *FORTE*, Lecture Notes in Computer Science, pages 21–40. Springer, 2018.

16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. `arXiv:2005.09520`.

17 Joseph M. Hellerstein et al. Serverless computing: One step forward, two steps back. In *CIDR*. www.cidrdb.org, 2019.

18 Daniel Hirschkoff, Enguerrand Prebet, and Davide Sangiorgi. On the representation of references in the pi-calculus. In *CONCUR'20*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2020.

19 Hans Hüttel et al. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

20 Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA):149:1–149:26, 2019. `doi:10.1145/3360575`.

21 Eric Jonas et al. Cloud programming simplified: A Berkeley view on serverless computing. Technical report, EECS Department, University of California, Berkeley, February 2019.

22 Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Charlton Barreto, and Yves Lafon. Web services choreography description language version 1.0, W3C candidate recommendation. Technical report, W3C, 2005. URL: `http://www.w3.org/TR/ws-cdl-10`.

23 Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

24 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

25 Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

26 Robin Milner. Functions as processes. *Math. Struct. in Computer Science*, 2(2):119–141, 1992.

27 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.

28 Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer, 1992.

29 Fabrizio Montesi. *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen, 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf.

30 Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with Jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, 2014. doi:10.1007/978-1-4614-7518-7_4.

31 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.

32 Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

33 Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Math. Struct. in Computer Science*, 6(5):409–454, 1996.

34 Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

35 Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520. IEEE, 2018. doi:10.1109/CVPR.2018.00474.

36 Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

37 Davide Sangiorgi. An investigation into functions as processes. In *Proc. Math. Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 1993.

38 Davide Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221:457–493, 1999.

39 Davide Sangiorgi. Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. *Theory and Practice of Object Systems*, 5(1):25–34, 1999.

40 Davide Sangiorgi. Termination of processes. *Math. Struct. in Computer Science*, 16(1):1–39, 2006.

41 Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.